

Aztec C Source Level Debugger for the Amiga

**Version 5.0
October 1989**

Copyright 1988,1989 by Manx Software Systems, Inc.
All Rights Reserved
Worldwide

DISTRIBUTED BY:

Manx Software Systems
P.O. Box 55
Shrewsbury, NJ 07702
(201) 542-2121

USE RESTRICTIONS

You are permitted to install and use this product on a single computer. Multiple CPU systems require supplementary licenses.

Before using any Aztec C products, the License Registration included with this product must be signed and mailed to:

Manx Software Systems
P.O. Box 55
Shrewsbury, NJ 07702

COPYRIGHT

This software package and document are copyrighted ©1988,1989 by Manx Software Systems. All rights reserved worldwide.

No part of this publication may be reproduced, transmitted, transcribed, stored in any retrieval system, or translated into any language without prior written permission of Manx Software Systems.

DISCLAIMER

Manx Software Systems makes no representations or warranties with respect to this product and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Manx Software Systems reserves the right to modify the programs and revise the contents of the manual without obligation to notify any person of such revision or changes.

TRADEMARKS

Aztec C, Manx AS, Manx LN, Z, and SDB are trademarks of Manx Software Systems. CP/M-86 and CP/M-80 are trademarks of Digital Research. MSDOS is a trademark of Microsoft. PCDOS is a trademark of IBM. UNIX is a trademark of AT&T Bell Laboratories. Macintosh and Apple II are trademarks of Apple Computer. Atari is a trademark of Atari Computers. Amiga is a trademark of Commodore-Amiga.

Manual Revision History

October 1989	Fifth Edition
December 1988	Fourth Edition
October 1988	Third Edition
July 1988	Second Edition
May 1988	First Edition

Table of Contents

Chapter 1 - Overview

Introduction	1 - 1
REQUIREMENTS	1 - 2
ABOUT THIS MANUAL	1 - 2
NOTATION CONVENTIONS	1 - 2
Technical Support	1 - 3

Chapter 2 - Getting Started

Introduction	2 - 1
Read the README File	2 - 1
Installation	2 - 2
Using the sdb Demonstration Programs	2 - 3
File Preparation	2 - 3
Startup	2 - 4
STARTING SDB	2 - 4
Where To Go From Here	2 - 6

Chapter 3 - Tutorial

Introduction	3 - 1
DISPLAYING SECTIONS OF THE SOURCE FILE	3 - 2
RUNNING THE PROGRAM	3 - 3
DISPLAYING THE TRACE OF CALLS	3 - 5
DISPLAYING VALUES AND COMPUTING EXPRESSIONS	3 - 6
WALKING UP AND DOWN THE FRAMES	3 - 7

DISPLAYING ASSEMBLY	3 - 7
---------------------------	-------

Chapter 4 - Commands

Overview	4 - 1
CATEGORIES	4 - 2
Basic	4 - 2
NAMES	4 - 3
Code and Data Symbols	4 - 3
Operator Usage of Names	4 - 3
LOADING PROGRAMS AND SYMBOLS	4 - 4
DEBUGGING LIBRARIES OR DEVICE DRIVERS	4 - 4
BREAKPOINTS	4 - 5
EXPRESSION BREAKPOINTS	4 - 6
TRACE MODE	4 - 6
BACKTRACING	4 - 7
MACROS	4 - 7
DISPLAYING SOURCE FILES	4 - 7
ABOUT WINDOWS IN SDB	4 - 8
Source and Data Displays	4 - 8
Command Display	4 - 9
Command Line Editing	4 - 9
Command Line History	4 - 10
Color Control	4 - 11
OTHER FEATURES	4 - 11
Use of Control Characters in a Command File	4 - 12
TERM DEFINITIONS	4 - 13
The Definition of expr	4 - 13
The Definition of addr	4 - 13
The Definition of RANGE	4 - 14

The Definition of CMDLIST	4 - 14
Detailed Description of Commands	4 - 14
AMIGA SPECIFIC COMMANDS	4 - 15
acc Show/set command window colors	4 - 15
acd Show/set data window colors	4 - 15
acs Show/set source windowcolors	4 - 15
LIST DISPLAY COMMANDS	4 - 16
add Display device list	4 - 16
adi Display interrupt list	4 - 16
adl Display library list	4 - 16
adp Display port list	4 - 16
adr Display resource list	4 - 16
am Display memory usage	4 - 17
ax Switch between main and alternate .dbg file	4 - 17
BREAKPOINT COMMANDS	4 - 17
bc Clear a single breakpoint	4 - 17
bC Clear all breakpoints	4 - 17
bd Display breakpoints	4 - 18
be Sets Expression Change Breakpoint	4 - 18
bm Sets Memory Change Breakpoint	4 - 19
br Resets breakpoint counters.	4 - 19
bs Set or modify a breakpoint	4 - 20
bt Toggle the call trace mode flag	4 - 20
bT Toggle the source line trace mode flag	4 - 20
DISPLAY COMMANDS	4 - 21
c Display Source Context	4 - 21
da Display local addresses	4 - 21
db Display memory in bytes	4 - 21
dw Display memory in words	4 - 21
dl Display memory in long words	4 - 21
d Display memory in last format	4 - 21
dc Display all code symbols	4 - 22

dd	Display all data symbols	4 - 22
df	Display source file lines	4 - 22
dg	Display global values	4 - 23
ds	Displays stack backtrace	4 - 24
dS	Displays stack, function arguments, and auto backtrace ..	4 - 24
"FIND SOURCE STRING" COMMAND		4 - 24
/	Finds string in source file	4 - 24
FRAME COMMANDS		4 - 25
fu	Frame up command	4 - 25
fd	Frame down command	4 - 25
GO COMMANDS		4 - 26
g	Execute the program	4 - 26
G	Execute the program, without setting table breakpoints ..	4 - 26
LOAD COMMANDS		4 - 27
ld	load device symbols	4 - 27
ll	load library symbols	4 - 27
lp	load program	4 - 27
Loading the Program		4 - 28
Loading the Symbol Table		4 - 28
MEMORY MODIFICATION COMMANDS		4 - 28
mb	Modify bytes of memory	4 - 28
mw	Modify words of memory	4 - 28
ml	Modify long words of memory	4 - 28
mc	Compare memory	4 - 29
mf	Fill memory with value	4 - 29
mm	Move memory	4 - 29
ms	Search memory	4 - 30
OUTPUT CONTROL COMMANDS		4 - 30
oe	Toggles command echo flag	4 - 30
op	Toggles pause flag	4 - 30
ow	Toggles switch to user screen	4 - 31
PRINT COMMAND		4 - 31

p	Generates formatted output	4 - 31
desc_code	List	4 - 35
Floating Point Number	Print Codes	4 - 36
Character	Print Codes	4 - 37
Special Purpose	Codes	4 - 37
P Command desc_codes	For Setting Current Address	4 - 38
QUIT COMMAND		4 - 39
q	Quit the debugger	4 - 39
REGISTER COMMAND		4 - 40
r	Register display	4 - 40
Description		4 - 40
SINGLE STEP COMMANDS		4 - 40
s	Single steps the program n times	4 - 40
S	Single steps into calls without display	4 - 40
t	Single steps n times stepping over calls	4 - 40
T	Single steps over calls, display last	4 - 40
UNASSEMBLE COMMANDS		4 - 41
u	Unassembles memory, with symbols	4 - 41
U	Unassembles memory, without symbols	4 - 41
MACRO COMMANDS		4 - 42
x	Create Macro Command	4 - 42
X	Display Macro Command	4 - 42
EXPRESSION COMMANDS		4 - 43
=	Display the value of an expression in several formats ...	4 - 43
e	Evaluate an expression	4 - 43
REDIRECT COMMAND INPUT/OUTPUT		4 - 43
<	Redirect command input	4 - 43
>	Log all I/O only	4 - 43
>>	Log commands only	4 - 43
HELP COMMAND		4 - 44
?	List command	4 - 44

CHANGE MODE COMMAND	4 - 44
z Change mode command	4 - 44
DELAY COMMAND	4 - 45
& Delay sdb	4 - 45
Command Summary	4 - 46
AMIGA SPECIFIC COMMANDS	4 - 46
BREAKPOINT COMMANDS	4 - 46
DISPLAY COMMANDS	4 - 46
FIND SOURCE STRING COMMAND	4 - 47
CHANGE FRAME COMMANDS	4 - 47
GO COMMAND	4 - 47
LOAD COMMAND	4 - 47
MEMORY MODIFICATION COMMANDS	4 - 47
OUTPUT CONTROL COMMANDS	4 - 47
FORMATTED PRINT COMMAND	4 - 47
QUIT COMMAND	4 - 47
REGISTER COMMAND	4 - 48
SINGLE STEP COMMANDS	4 - 48
UNASSEMBLY COMMANDS	4 - 48
MACRO COMMANDS	4 - 48
EXPRESSION COMMANDS	4 - 48
REDIRECT COMMAND INPUT	4 - 48
HELP COMMAND	4 - 48
CHANGE MODE COMMAND	4 - 48
DELAY COMMAND	4 - 48

OVERVIEW

GETTING STARTED

TUTORIAL

COMMANDS

OVERVIEW

1

Chapter 1 - Overview

Introduction

Manx **sdb** is a fast, easy to use, interactive source level debugger. You will appreciate the simplicity of working with **sdb**. If you are an experienced programmer, you will recognize the time and effort you save debugging with **sdb**.

Use **sdb** to debug programs created with the Aztec C Compiler. The windows in **sdb** display C source and command output separately, with a third window for entering commands.

With **sdb**, you can...

- ...debug programs at the C source and assembly language levels*
- ...display all active function names*
- ...display values of passed parameters*
- ...examine variables from any active function*
- ...use function or line-by-line tracing*
- ...set breakpoints by lines, functions, or variables*
- ...see actual C source as it executes*
- ...customize the debugging environment with reusable command macros and procedures*
- ...and more!*

REQUIREMENTS

Manx recommends a minimum of 256K bytes of available RAM memory for use with **sdb**; the debugger uses about 180K. **sdb** runs under the CLI. **sdb** can be used on 68000, 68010, 68020, and 68030 systems.

ABOUT THIS MANUAL

This manual describes how to use the Source Level Debugger and is designed to be used primarily as a reference. Though this manual does include a brief tutorial, the best way to learn, in depth, how to use **sdb** is to work through the demonstration programs that have been included on your **sdb** disk. See the `readme` file or Chapter 2 of this manual for instructions on how to execute the demonstration programs.

Throughout this manual, we assume that you already have a basic understanding of how to use Aztec C. If you need additional technical information about using Aztec C, refer to your Aztec C documentation.

This manual is divided into the following sections:

Introduction	Describes the requirements and features of sdb .
Getting Started	Gives the procedures necessary to install sdb , start up sdb , and run the sdb demonstration program.
Tutorial	Gives easy-to-follow directions for using sdb .
Commands	Describes the features of sdb and how to use them. Defines some sdb terms and describes the sdb commands in detail. Includes a command summary for easy reference.

NOTATION CONVENTIONS

Throughout this manual, we use the following conventions:

input	to indicate data entered by the user (e.g., commands, options, and functions)
--------------	---

output	to show text that is generated by the computer
DEFINITION	small uppercase bold is used on terms that may be new to the user; most likely will include explanation or definition of term
[choice1 choice2]	braces and a vertical bar mean that you have a choice between two or more items
<i>placeholders</i>	information that must be supplied by the user, for example, <i>filename</i> , <i>range</i> , <i>identifier</i> , etc.
[<i>optional</i>]	to show optional information

Technical Support

At Manx, we build dependability into our products. However, if you should find that you need help, rest assured that we provide it.

Refer to your *Aztec C User Guide* for information on the Technical Support that Manx provides. If you have any problems with **sdb**, follow the procedures outlined in **Technical Information** chapter so that we may best be able to assist you.

GETTING STARTED

2

Chapter 2 - Getting Started

Introduction

This chapter takes you through the steps necessary for getting started with **sdb**. Before you begin using **sdb**, you should:

- Read the **readme** file.
- Install **sdb**, as described in the **Installation** section of this chapter, on your hard drive OR create an **sdb** "work diskette"
- Run the **sdb** demonstration programs that have been included on your disk.
- Prepare your files for debugging.
- Startup **sdb**.

This chapter describes each of these procedures in detail.

Read the README File

If there is a **readme** file on your **sdb** disk, you can access it by entering "**type readme**" from the CLI. The **readme** documents provide information that

may not be in the manual or may be more recent than information in the manual.

If you want to send the **readme** document to the printer, use the command:

```
type > PRT:  readme
```

Installation

Once you have verified that your files are complete, you should load **sdb** onto your hard disk or create a working floppy disk.

To load **sdb** onto your hard disk:

1. Turn on your machine and enter the CLI.
2. Insert your **sdb** disk into your floppy drive. Change directory to your hard disk's **bin** directory. Copy the file **sdb** into your bin directory using the command:

```
copy df0:sdb  sdb
```

3. Change directory back to the root directory of your hard disk. Create a scratch directory and load the **sdb** examples into the scratch directory.

To create an **sdb** "work disk":

1. Turn on your machine and enter the CLI. Do not boot off of your Aztec C disk, as it does not give you access to CLI.
2. With the **sdb** master in drive zero, insert your work disk into drive one.
3. Copy the contents of your SDB disk to the work disk using the command

```
diskcopy df0: df1:
```

Once you have run through the tutorial and the examples, you may delete all but the **sdb** file from your work disk. You do not need any other files from the disk in order to use **sdb**.

Using the sdb Demonstration Programs

As explained in Chapter 1, the **Overview**, the best way to learn how to use **sdb** is to work through the demonstration programs that have been included on your **sdb** disk. After you have worked through the demonstration programs, you should also read the written tutorial, Chapter 3, which is included in this manual.

To start the demonstration, boot your machine into CLI. Do not boot from the SDB disk, as it does not make CLI available to you. As explained above, under "Read the **readme** File", it is a good idea to print out the **readme** file before beginning work. To begin the demonstration, change directory to the **demo** directory by typing:

```
cd demo
```

Then, simply type **demo**. After making sure that you want to proceed, the demonstration will "walk you through" the basics of using **sdb**. For a complete understanding of the power and flexibility available with **sdb**, you should work through each of the demonstrations included.

File Preparation

sdb works on executable files. To use **sdb**, however, you must create a debug file as well. The following steps will show you how this is done:

1. Create a C source file.
2. Compile using the **-bs** option, as shown:

```
cc -bs filename.c
```

3. Link using the **-g** option, as shown:

```
ln -g filename.o -lc
```

You will now have two files: an executable file and a debug file. The debug file will have the extension **.dbg**.

Note: To use **sdb**, you must have version 3.6 or higher of Aztec C.

Startup

You can start **sdb** in one of two ways:

1. From the CLI, by typing

```
sdb filename
```

2. From the CLI by invoking **sdb** and then using the **lp** command to load the program. (See the section on the **lp** command in the **Commands** chapter.)

The following sections discuss these startup methods.

STARTING SDB

When you start **sdb** from the CLI, as shown in step 1 above, you may also specify certain options and parameters, in the form of:

```
sdb [options] filename [parameters]
```

where

[*options*] are any one of the following:

- | | |
|----------------------|--|
| -a | starts sdb in assembly mode (default is C source mode; see z command in the Commands chapter.) |
| -cco,t,b,p | set command area colors(<i>o</i> =outline, <i>t</i> = text, <i>b</i> =background, <i>p</i> =prompt). For each letter in the - cc , - cd , and - cs options, enter one of the numbers 0 through 3 to select one of the four workbench colors. |
| -cdt,b | set the data area display (<i>t</i> =text and <i>b</i> = background) |
| -cst,b,ht,bt. | set the source display colors (<i>t</i> =text, <i>b</i> =background, <i>ht</i> = highlighted text, and <i>bt</i> =high-lighted background) |

-cwl,t,w,h	controls the positioning and size of the sdb window. All four numbers must be present; they determine (in order) left edge, top edge, width, and height.
-d#	specifies data window buffer size with valid range of $512 < \# < 32000$ (default is 4096 bytes).
-e	toggles echoing of commands to the data window (see the oe command in the Commands chapter.)
-h#	specifies history window buffer size with valid range of $128 < \# < 10240$ (default is 512 bytes).
-mfile	forces redirection to an input <i>file</i>
-n	does not look for <i>file.mac</i>
-p	sets prompt for -w mode
-r	use this option if the program being debugged has been made resident using the REZ utility.
-spath	path is a string to be prefixed to source files upon opening
-w	disables the windowing part of the debugger display

and *[parameters]*, like *arg1*, *arg2*, ..., are character strings to be passed to the program. When invoked, **sdb** loads the program to be debugged, starts it, and then stops at the entry point to the main function displaying that line.

The startup options can be changed and made somewhat permanent by setting the environment variable **SDBOPT** to the appropriate values. For example, to preset the colors for the three windows, enter the following:

```
set "SDBOPT=-cs1,3,2,0 -cd3,2 -cc0,0,1,2"
```

If the program *filename* specifies a drive or directory, **sdb** searches for the program file in only that particular area. Otherwise, it searches the current directory on the default drive.

Arguments are passed to the program using the **argv** parameters of the program's main function: **arg1** is pointed at by **argv[1]**, **arg2** **argv[2]**, and so on. **argv[0]** contains the name of the program.

-spath creates the **fpath** macro, which tells **sdb** to look for source files in a sequence of directories. By default, **sdb** looks in the current directory. If you do not specify another directory or subdirectory, **sdb** assumes the current directory should be searched. However, you may prefer to separate your executable files from your source files. Therefore, **sdb** searches multiple directories as long as you specify the directories or subdirectories on the command line, separated by a ";" or "!". For example,

```
-sdir1;vol:dir2;...
```

or

```
-sdir1!vol:dir2!...
```

You may also invoke **sdb** by typing **sdb** with no arguments. You must use the **lp** command to load the program you want to debug.

Where To Go From Here

Now that you have worked through the steps necessary for getting started with **sdb**, you should go on to the **Tutorial**, Chapter 3.

Chapter 3 - Tutorial

Introduction

This tutorial introduces you to the Source Level Debugger. As explained in the previous chapter, **sdb** works on executable files. To use **sdb**, however, you must create a debug file as well. Refer to the **File Preparation** section of Chapter 2, **Getting Started**, for instructions on how this is done.

With this tutorial, we will assume that you already have a basic understanding of the Aztec C software development system and how it is used. If you have any questions about using Aztec C, refer to your Aztec C documentation.

When using the debugger, you should make use of the help screens, which can be obtained by entering

?

Further detail is provided for many of the commands by entering the first letter of the command followed by a question mark. For example, typing

f?

will display information on the commands **fu** and **fd**.

To begin this session, you must invoke **sdb**. Refer to the **Startup** section of Chapter 2, **Getting Started**, for instructions on how this is done. As explained in that section, you may enter **sdb** from the CLI.

When invoked, **sdb** loads the program to be debugged, starts it, and then stops at the entry point to the main function, displaying that line.

DISPLAYING SECTIONS OF THE SOURCE FILE

Two commands, **c** and **df**, are provided for displaying your source file.

c is used to display the current source line. It takes no arguments.

df can be used to display lines other than the current source line. The format of the **df** command is:

df *[filename]* *range*

where *filename* is the name of the file to be displayed and *range* is the line number that will be displayed at the beginning of the source window. Note that *filename* is optional and defaults to the current file if none is specified.

range is one of the following:

line

line ... line

line,count

line and *count* are expressions yielding an integer result. If *line* is specified alone, the line indicated is displayed at the beginning of the source window.

The **c** command is very useful if you have used **df** to display another file or have scrolled out of the source window. If this is the case, then typing

c

will bring you back to your current file at the current line.

As with all display commands, pressing <CR> after issuing the command causes the next set of lines of source to be displayed.

RUNNING THE PROGRAM

Once the program is loaded, debugging consists of setting breakpoints and running the program. Breakpoints can be one-time breakpoints or can be set as permanent breakpoints.

If you do not wish to set permanent breakpoints, you can control execution of the program using the single step and `go` commands.

The format of a single step command is:

`[count]s`

`[count]S`

`[count]t`

`[count]T`

where *count* is a positive integer that causes the debugger to single step *count* times, printing information for each breakpoint before stopping.

If *count* is zero, single stepping is performed until a breakpoint is hit, an error occurs, or the program finishes.

The difference between *s* and *t* is that *s* single steps *into* calls while *t* single steps *across* calls. In effect, *t* treats a call as a single line.

S and *T* step *count* times, displaying information only for the last breakpoint taken. *S* steps into calls just like *s* while *T* treats calls as a single line.

In source level mode (the default), single step is by source line. In assembly mode, (set by typing `z<CR>`) single step is by machine code instruction.

If single stepping encounters a function for which no source line information is available (a library function for example), single stepping into the function causes the debugger to step over the call just as it would with *t*.

To set a permanent breakpoint, type:

[count]bs *addr*[:*command*]

where *count* is an integer expression, *addr* is:

[*filename*].*line*

***function*[.*line*]**

addressexpression

and *command* is a set of debugger commands separated by semicolons.

For example, you can enter:

bs linkmain.c.39

This sets a breakpoint on line 39 of `linkmain.c`. Alternatively,

bs getfld.10

sets a breakpoint at line 10 in the function `getfld`.

bs getfld+10

sets a breakpoint at 10 bytes beyond the start of `getfld`. Entering

bs .10

will set the breakpoint at line 10 in the file currently being displayed.

Finally, to make the program go, you simply type `g<CR>`. The program executes until it encounters a breakpoint or terminates.

The format of the go command is:

g [@] [*addr*]

where @ means "go until a return after *addr* is found" and *addr* is as described above. *addr* is set as a temporary breakpoint and will not be remembered after execution of the command.

For example:

g @

means go until the current function returns. And

g linkmain.c.39

means go to line 39 in linkmain.c

DISPLAYING THE TRACE OF CALLS

It is often useful to know where the program is in a set of nested calls and what the arguments and local variables are to each of the calls. To display the nested stack of calls in `sdb`, simply type

ds

Each of the calls currently active is displayed, together with its arguments; each argument is displayed in a form appropriate to its type.

Type

ds

and the functions are displayed with the types, names, and values of each function argument, and auto variables.

For example:

ds

might cause the following display

```
_main(int argc=1, char ** argv = 0x00C051B8)
__main (0x0000, 0x0002, 0x00C0, 0x5040
.begin ()
```

while

ds

could cause:

```
_main(int argc = 1, char **argv = 0x00C051B8)
    int i = 0
    long j = -1
    char name[8] = "hello."
__main()
.begin
```

DISPLAYING VALUES AND COMPUTING EXPRESSIONS

sdb provides some simple but powerful facilities for displaying variables, arrays, and structures. These are the **p** (print) and the **e** (evaluate) commands. For example, to print a structure named **symbol**, you simply enter:

```
p symbol
```

which might result in:

```
struct symboltb symbol = {
    int s_flag = 10
    char *s_name = 0xFF42
    int s_value[2] = {
        10,5
    }
}
```

Suppose then you want to display the string pointed to by **symbol.s_name**. You simply type:

```
ps *symbol.s_name
```

The **s** following the **p** command indicates that you are overriding the default format for pointer to **char** and the **s** indicates that the new format is string.

So the result might be:

```
pointer
```

In addition to the print command, the debugger has an evaluate command so that you can perform general C expression evaluation, including calls to C functions, assignment, pre- and post-increment and decrement, casts, and conditionals.

```
e c = getchar()
```

might result in:

=10

WALKING UP AND DOWN THE FRAMES

In using the expressions shown above, you are generally limited to referring to variables that are visible by C rules at the point where execution stopped. Therefore, you cannot refer to local variables or functions that are not active or are not the “current” function. (Statics, however, may be referred to by qualifying them with the name of the file or function they were declared in, e.g. `linkmain.c.name` or `main.name`.)

In order to refer to names in other active functions, you can change `sdb`’s notion of what the current function is by walking up and down the call frames, using the commands `fu` for frame up, and `fd` for frame down. These commands walk up the call frames, displaying the line from which the next frame down was called and making visible all of the current call’s local variables.

DISPLAYING ASSEMBLY

Finally, you can display the assembly code at any address with the `unassemble` command. Its format is:

`u addr`

`U addr`

where *addr* is as described above.

`u` does a disassembly with symbols substituted where possible for global and local variables. **`U`** disassembles without symbol substitution but with the hex for the code shown as well as the assembly.

Chapter 4 - Commands

Overview

sdb commands consist of one, two, or three characters. The first identifies the command category and the second and third identify the specific operation to be performed. If the command name contains only one character, it is the only command in a category.

If you are using Motorola Fast Floating Point, use **sdbf**. **sdbf** is identical to **sdb** except for the support of Motorola fast float.

This chapter contains the following:

Categories	describes the general command categories
Detailed Description	describes how to use sdb and sdbf commands in detail
Command Summary	lists the commands with a brief description.

CATEGORIES

Basic

sdb has two types of commands for examining memory: display (**d**) and print (**p**). The display commands **db**, **dw**, and **dl** simply display hexadecimal bytes, words, and long words, respectively.

The print command, **p**, is more powerful, allowing you to print variables, arrays, and structures by name without having to worry about data types. For example, you can tell **sdb** to print a structure whose name is **symbol** by typing

```
p symbol
```

sdb figures out the data types and prints the structure and its members in the proper format. You can also display strings that are pointed to by a structure member using the **p** command.

The evaluate command, **e**, allows you to perform general C expression evaluation, including calls to C functions, assignment, pre- and post-increment and decrement, casts, and conditionals.

The register command, **r**, displays and modifies the 68000 registers.

The frame commands, **f**, allow you to walk up and down the call frame.

The memory modify commands, **m**, allow you to modify memory.

The unassemble commands, **u**, display code symbolically, in a form similar to its appearance in an assembly language source file.

The **s**, **t**, and **g** commands cause your program to be executed. **s** and **t** commands “single step” through your program, e.g., they execute a specified number of instructions in your program and then return control to **sdb**.

The **g** commands transfer control of the processor unconditionally to your program. In this case, **sdb** regains control when your program terminates, when an error occurs (such as division by zero), or when a “breakpoint” is taken. (Breakpoints are described in more detail later in this chapter.)

The help command, **?**, displays a summary of all **sdb** commands. For some command categories, you can get information about the commands in a category by typing the first letter of the category followed by a **?**. For example, type

```
m?
```

to get information about the memory modification commands (all of which contain a first letter **m**).

NAMES

sdb allows memory locations to be referenced by name as well as by location. It learns a program's variable names by reading the file containing the program symbol table.

The Linker generates a symbol table file for a program in response to option **-g**. There are various ways for **sdb** to read the symbol table and these are described below.

sdb allows global symbols, automatic variables, static variables, arrays, and structures to be accessed by name.

Code and Data Symbols

sdb classifies symbols as being either code or data symbols, e.g., they refer to a location in a physical code segment or a physical data segment.

There are two commands for viewing the symbols that are known to **sdb**: **dc** and **dd**, which display code and data symbols, respectively.

Operator Usage of Names

When a C source program is compiled, all global symbols are truncated to a maximum of 31 characters and are then prepended with an underscore character.

To refer to symbols which, in a C source file, contain less than 31 characters, the inclusion of the prepended underscore character is optional. If it is specified, **sdb** searches for that symbol in its symbol table. Otherwise, it first searches for the specified symbol. If the search fails, it prepends an underscore to the name and searches again. To refer to symbols that contain 32 or more characters, only the first 31 are significant. **sdb** ignores all other characters in the name.

LOADING PROGRAMS AND SYMBOLS

A program and its symbols can be loaded into memory when **sdb** is started. In this case, the command line defines the program to be loaded.

The **sdb** load program command, **lp**, can also be used. If **sdb** is started with a program name specified on the command line, the **lp** command can

LOADING PROGRAMS AND SYMBOLS

A program and its symbols can be loaded into memory when **sdb** is started. In this case, the command line defines the program to be loaded.

The **sdb** load program command, **lp**, can also be used. If **sdb** is started with a program name specified on the command line, the **lp** command can be used to reload the same program for another debugging session. If **sdb** is started without specifying a program name, the **lp** command must be issued with the desired program name. Only one user program can be in memory at once. When told to load a program, **sdb** automatically tries to load the program symbol table too. It assumes the symbol table file has the same name as the program file, with the extension changed to **.dbg**.

When a program exits, it must be reloaded with the **lp** command before execution can begin again.

DEBUGGING LIBRARIES OR DEVICE DRIVERS

Because a library or device driver is not “run” in the same way as a program, **sdb** contains special support to enable these to be debugged at the source level. First, they must be compiled with option **-bs** and linked with option **-g**.

Then the binary generated by the linker should be placed in the **libs:** directory while the **.dbg** file should be in the source directory. Next the program that will make calls to the library or device must also be compiled and linked to generate source level information.

Now, **sdb** should be invoked on the test program. When the program is loaded, it should be run far enough so that the appropriate library or device is in memory. Use the **adl** or **add** command to verify this. Then the **ll** or **ld** command should be used with the appropriate library or device name as in:

```
ll mylib
```

or

```
ld mydev
```

BREAKPOINTS

Before transferring control of the processor to your program in response to a **g** command, **sdb** can set **BREAKPOINTS** at specified locations in the code. When your program reaches a breakpoint, **sdb** regains control.

A breakpoint has a skip count associated with it that allows it to be passed several times before actually taking the breakpoint and returning control to **sdb** and the user. When a breakpoint is reached, **sdb** is always activated. It increments a counter associated with the breakpoint. When the counter's value is greater than the breakpoint's skip count, the breakpoint is taken. For example, **sdb** retains control of the processor. Otherwise, **sdb** returns control of the processor to your program after the breakpoint. By default, a breakpoint's skip count is 0. Thus, each time the breakpoint is reached, it is taken.

A breakpoint can also have a sequence of **sdb** commands associated with it. When a breakpoint is taken, these commands are executed before **sdb** allows you to enter commands. For example, if you want only to examine a variable each time a certain location in the code is reached and then have the program continue execution, you could define a breakpoint at the location and specify a list of commands to do just that. The first command in the sequence would be a **d** command to display memory, and the second would be a **g** command to continue execution of the program.

There are two ways to define breakpoints: with the **g** command and with special breakpoint commands, whose first letter is **b**.

The breakpoint commands manipulate a table of breakpoints: There are commands for entering breakpoints into the table, displaying the entries, resetting their counters, and removing them from the table.

There is a difference between a breakpoint defined in a **g** command and those in the breakpoint table: The **g** command breakpoint is temporary, while a breakpoint table is more permanent (it exists until removed from the table). Before transferring control to your program in response to a **g** command, **sdb** sets all breakpoints that are in the breakpoint table and that are specified in the **g** command itself. When a breakpoint is taken, **sdb** removes all breakpoints from the code and forgets all about the **g** command breakpoint. The breakpoint table breakpoints, however, are still in the table and will be set back in memory when control is again returned to your program.

sdb remembers the skip counter associated with a breakpoint in the breakpoint table. When it sets breakpoints in memory, the count for such a breakpoint is set to its remembered value (e.g., its value in the table). When a breakpoint is taken, the accumulated count for the breakpoints in memory is saved in the breakpoint table.

EXPRESSION BREAKPOINTS

The breakpoints described above are taken when a program reaches a specified point in the code. A second type of breakpoint, called an expression breakpoint, is taken when a specified expression evaluates as true.

With an expression breakpoint set, **sdb** detects the instruction that causes the specified expression to evaluate as true, assuming your program is being single stepped using an **s** command.

When an **s** command is used to single step a program and an expression breakpoint is set, **sdb** evaluates the specified expression after each instruction is executed and takes a breakpoint when appropriate.

The **be** command is used to set and remove expression breakpoints.

TRACE MODE

sdb supports two trace modes that display information whenever a function is entered or exited or when a source line is passed.

With the first mode enabled upon entry to, and exit from, a function, the function name, its arguments, and return values are displayed.

The commands **bt** and **bT** affect trace mode: **bt** enables and disables call trace mode, and **bT** enables and disables source line trace mode.

BACKTRACING

When **sdb** regains control from an executing program (for example, because a breakpoint was taken), it has the ability to display information on how the program got to its current location: The **ds** and **dS** commands display information about the currently executing function, and the func-

tion that called it, and so on, back to the `Manx` function `__main()`, which called your function `main()`.

For each function, `ds` displays its name, arguments that were passed to it, and the address to which it will return. `ds` displays the function's automatic variables as well.

MACROS

`sdb` allows you to define and execute macros, e.g., a sequence of `sdb` commands.

Macros are written to a file that is saved so that macros can be reused in a subsequent session. The filename for the saved macros is derived by taking the program name and appending a `.mac`. The `sdb` command `x` is used both to define and execute a macro.

DISPLAYING SOURCE FILES

Since `sdb` is a source level debugger, it allows you to display source files, thus providing a convenient means to examine the source of a program being debugged.

Only a single source file can be examined at a time. The display source command, `df`, and the context command, `c`, can be used to display its lines.

The find string command, `/`, finds a character string in the source file.

ABOUT WINDOWS IN SDB

`sdb` provides a single independent window created in the Workbench screen. The title on the window is "Aztec SDB:", followed by the name of the source file being displayed and the current function name in which the debugger has stopped.

Within this window are three independent display areas: a source area, a command area, and a data area. The entire window can be resized using the sizing icon located in the lower right corner of the window. The minimum size guarantees two source lines, the command line, and three

data lines. Lines that are too long to fit within the source or data displays are truncated, which is indicated by a \$ character in the last column.

Source and Data Displays

Both the source and data areas operate similarly. Each can be considered to be a window containing text or data from a larger set of text or data. The position of this window on the larger set is controlled by an icon at the right side of the display. This gadget consists of a rectangle with a smaller square within it, bordered at the top and the bottom with two arrows pointing up and down, respectively. The arrows are used for fine adjustments of the display, while the squares are used for larger adjustments.

The outer rectangle of the gadget represents the total amount of text or data that can be displayed. The inner square, by its position, represents the position of the displayed text within the total. To reposition the window, move the mouse pointer over the scroll box and move the inner square up or down, while holding down the mouse button. As the square is moved, the display will be updated with the contents appropriate to the new position. When the desired position is reached, release the mouse button.

To move the display only one display's worth at a time, point the mouse either above or below the inner square but within the outer rectangle. Now, each mouse click will move the display up or down by the number of lines within the display.

To move the display one line at a time, point the mouse at one of the two arrows and click the mouse button. The display scrolls for each click of the mouse button. It scrolls continuously if you hold the mouse button down.

The displays may also be scrolled using the UP and DOWN cursor keys on the keyboard. To move the source display, hold down either SHIFT key while pressing the UP or DOWN cursor keys. The data display is similarly controlled by using the ALT key instead of the SHIFT key.

The source display is a window on the current text file. The data display is a circular buffer of 2000 characters. As lines are added to the end of the display, lines are removed from the beginning of the display. Because the buffer is character based, the display can show a few long lines, many short lines, or something in between.

Command Display

The command display is a single line display that consists of a **CMD?** prompt, an editing area, and a small UP/DOWN gadget to the extreme right.

The UP/DOWN gadget is used to position the command display within the **sdb** window. When the mouse is pointed at the gadget and the button is held down, an outline of the command area is displayed that follows the position of the mouse pointer. When the button is released, the command line is moved to the new position. This automatically causes the sizes of the source and data displays to be adjusted, and their contents and controls are redisplayed to reflect the new sizes.

The command display can also be moved by holding down the **SHIFT** and **ALT** keys simultaneously while pressing the **UP** and **DOWN** arrow keys.

Command Line Editing

When the command display is awaiting input, a block cursor is displayed within the editing area of the command display. At this point, characters may be typed into the display area using the keyboard. As each key is typed, it is inserted and the cursor moves to the right. When the **BACKSPACE** key is pressed, the cursor moves left one character and that character is deleted. A line can be up to 128 characters in length. If the line exceeds the width of the display, it will automatically scroll to the left as characters are added. When a line is finished, pressing the return key causes the line to be parsed and executed by **sdb**.

In addition to the **BACKSPACE** key, there are a number of editing features that are available within the command display. First, there are a number of ways to change the cursor position. Use the **LEFT** and **RIGHT** arrows to move the cursor one character position in the corresponding direction. Hold down either **SHIFT** key and press the **LEFT** or **RIGHT** arrow key to move the cursor to the beginning or end of the line. And point the mouse anywhere in the line and press the mouse button to move the cursor to that spot.

Once the cursor is positioned, characters can be deleted at the current position by using the **DEL** key. New characters can be inserted by simply typing them. Space is made for each character as it is typed.

If instead of inserting characters, the new characters are to replace the existing characters, press ESC, to switch from insert mode to replace mode. Press ESC again to switch back to insert mode.

Command Line History

When the RETURN key is pressed to signal the end of an input line, the line is added to the end of a 512-byte buffer. Longer lines mean fewer lines in the buffer, while shorter lines mean more lines in the buffer.

When a line is to be added to the buffer, it is first searched to see if the same exact line already exists in the buffer. If not, then the new line is simply added to the end and lines from the beginning are removed to make space. If it is found, the old line is simply moved to the end of the buffer after moving all the intervening lines. In this case, no lines are removed from the beginning. This is done so that simply typing `s` followed by RETURN a hundred times while stepping through a program will not flush all the old lines out of the buffer.

For each press of the UP key, the command line shows the previous line in the history buffer. When the desired line is found, it can be entered by pressing RETURN immediately or it can be edited as discussed in the previous section.

Use these additional character key sequences to move as follows:

- F10** to swap screens between your program and `sdb`
- ESC** to toggle insert versus overstrike mode
- CTRL-C** to terminate output
- CTRL-X** to clear the command line

Color Control

Since the `sdb` window is created in the Workbench screen, `sdb` is limited to the four Workbench colors selected using the Preferences program. However, there is full control of the colors used within each display area. The Workbench colors are referred to by numbers 0 to 3. These correspond to the four colors displayed in order by the Preferences program.

The data display has the simplest choice of colors, namely the color used to display text and the color of the background. The default is 1,2 or text set to color 1 and background set to color 2.

The source display has four choices of colors. The first two correspond to the color and background of text similar to the data display. The second two correspond to the color and background of highlighted text. Highlighted text is used to show the current position of the program within the file. The default is 1,0,1,3 or text set to color 1, background set to color 0, highlighted text set to color 1 and highlighted background set to color 3. To display highlighted text as the inverse of regular text, simply reverse the text and background values as in 1,0,0,1.

Finally, the command display area has four choices. Just like the source and data displays, the text and background can be chosen. In addition, the text color of the CMD? prompt can be chosen as well as the color of the outline of the command area. The outline consists of a one pixel high line above and below the command area. The default is 1,1,0,1 or an outline in color 1, text in color 1 with a background of color 0, and the prompt set to color 1 as well.

The display colors can be examined and changed by using the **ac** commands in **sdb**. In addition, the colors can also be set using the **-cs**, **-cc**, and **-cd** options to **sdb**. To make the changes somewhat automatic, the environment variable **SDBOPT** is scanned for arguments in addition to those used to invoke **sdb**.

OTHER FEATURES

Some other features of **sdb** that have not yet been discussed are:

- redirect **stdin** command, **<**, causes **sdb** to read commands from a specified device or file and then continue reading commands from the console. Log commands, **>**, and **>>**, allow the logging of either all I/O or the commands only to a separate file.
- evaluate expression command, **=**, evaluates an expression.
- help command, **?**, lists commands with a brief description of each.

Use of Control Characters in a Command File

Most of the keyboard commands have control character equivalents. These may be used directly from keyboards that support the use of control characters, but their primary use is for command files. This can be seen in the command file scripts used to run the `sdb` demos found on your disk. The following are the control characters and their keyboard equivalents.

Control Character	Keyboard Sequence	Action Performed
^K	up arrow	scroll back in history buffer
^D	down arrow	scroll forward in history buffer
^G	left arrow	move cursor left
^L	right arrow	move cursor right
^T	shift - up arrow	scroll C source down
^U	shift - down arrow	scroll C source up
^V	shift - left arrow	move cursor to far left
^W	shift - right arrow	move cursor to far right
^N	ALT - up arrow	scroll data down
^D	ALT - down arrow	scroll data up
^[	F10	view application screen
^Y	ESC	toggle between Insert and Replace mode

TERM DEFINITIONS

This section defines some terms that are used in the command descriptions.

These terms are *expr*, *term*, *addr*, *range*, and *cmdlist*.

The Definition of *expr*

An *expr* is any valid C expression.

For example, an *expr* can consist of a single *term*, a series of *terms* separated by operators, the use of registers by their standard names, or 32-bit values representing memory locations.

Here is an example of *expr*:

```
si
x + 2.0
(i == j) ? 2 : 3
0xC0B54
sin(y)
array[j]
```

The Definition of *addr*

An *addr* is the name of a C variable, an address constant, or a C expression that yields an address. Examples of an *addr* are as follows:

<code>linkmain.c.19</code>	address of the code corresponding to line 19 of <code>linkmain.c</code>
<code>.19</code>	same as above if current file is <code>linkmain.c</code>
<code>main</code>	address of <code>main</code>
<code>0x1532</code>	an address at <code>0x1532</code>
<code>token[j]</code>	address of <code>j</code> th element of array <code>token</code>

The Definition of RANGE

A *range* defines a block of memory. It has one of the following forms:

```
addr,cnt
addr to addr      (the word "to" must be included)
addr
,cnt
```

The form *addr,cnt* specifies the starting address, *addr*, and a number, *cnt*. *cnt* is interpreted differently by different commands. For example, the disas-

semble code command, **u**, displays *cnt* lines, while the display bytes command, **sdb**, displays *cnt* bytes.

The form *addr to addr* specifies the starting and ending addresses of the range. A full range need not be explicitly specified, because **sdb** remembers the last-used range and sets unspecified RANGE parameters from the remembered values:

- When a *range* is specified that consists of a single *addr*, the last used *cnt* is used.
- When a *range* is specified that consists of *cnt*, the next consecutive address is used, and the remembered count is changed to the new value.
- When nothing is specified as the *range*, the next consecutive address is used as the starting *addr*, and the *cnt* is set to the remembered value.

The Definition of CMDLIST

A *cmdlist* is a list of commands. It consists of a sequence of commands or macros separated by semicolons:

command [*xcommand* ...]

If a macro is in a *cmdlist*, it must be the last command in the list.

Detailed Description of Commands

The commands are listed alphabetically within categories.

AMIGA SPECIFIC COMMANDS

- **acc** Show/set command window colors
- **acd** Show/set data window colors
- **acs** Show/set source window colors

Syntax

```
acc [out, text, back, prompt]
acd [text, back]
acs [text, back, high, back]
```

Description

The **acc**, **acd**, and **acs** commands control the color for the command data and source displays, respectively. See the **Overview** section of this manual for a detailed description of "windows" and "color codes." Because **sdb** is created in the Workbench screen, it is limited to the four Workbench colors selected using the Preferences program. Choose the four Workbench colors by referencing numbers 0 through 3.

acc: You have four color choices for the command display—text, background, prompt text, and command area outline. The default values are:

1, 1, 0, 1

where

Outline	Color 1
Text	Color 1
Background	Color 0
Prompt Text	Color 1

acd: You have two color choices for the data display: text and background. The default values are:

1, 2

where

Text	Color 1
Background	Color 2

acs: You have four color choices for the source display: the first two correspond to the color and background of text, the second two correspond to the color and background of highlighted text. (Highlighted text is used to show the current position of the program within the file.) The default values are:

1, 0, 1, 3

where

Text	Color 1
Background	Color 0
Highlighted Text	Color 1
Highlighted Background	Color 3

LIST DISPLAY COMMANDS

- **add** Display device list
- **adi** Display interrupt list
- **adl** Display library list
- **adp** Display port list
- **adr** Display resource list

Syntax

add
adi
adl
adp
adr

Description

These commands display the addresses and names of one of the various lists pointed to by ExecBase in the Amiga.

- **am** Display memory usage

Syntax

am

Description

This command displays the amount of free memory in the system.

- **ax** Switch between main and alternate .dbg file

Syntax

ax

Description

This command is used to switch back and forth between the two “personalities” of **sdb**—that which knows about the source and symbols of the test program discussed earlier in this **Command** section, and that which knows about the source and symbols of the library or device driver.

BREAKPOINT COMMANDS

- **bc** Clear a single breakpoint
- **bC** Clear all breakpoints

Syntax

bc *addr*
bC

Description

These commands delete breakpoints from the breakpoint table. **bc** deletes the single breakpoint specified by the address *addr* and **bC** deletes all breakpoints from the table.

- **bd** Display breakpoints

Syntax:

bd

Description

This command displays all entries in the breakpoint table.

For each breakpoint, the following information is displayed:

- Its address, using a symbolic name, if possible
- The number of times it has been "hit" without a breakpoint being taken
- The skip count for it
- The command list for it, if any.

For example, a **bd** display might be:

# - Address	hits	skip	command
C1 - c62f9c _printf	1	2	
2 - c62b28 prog.c.19	0	0	p i

In this example, two breakpoints are in the table. The first is at the beginning of the function **_printf**; a breakpoint will be taken for it every third time it is reached, and no command will be executed. Given its current hit count, a breakpoint will be taken the next time **_printf** is reached.

The second is on line **19** of the file **prog.c**; a breakpoint will be taken each time this line is about to be executed and will print the value of the variable **"i"**.

- **be** Sets Expression Change Breakpoint

Syntax

be [*expr*]

Description

This command is used to set and clear an expression breakpoint. This breakpoint is set by entering an arbitrary C expression. This expression is then evaluated each time a function is entered or exited. Whenever it is true, the breakpoint is taken, otherwise not. To cause the expression to be evaluated on each C source line, you must be running in single step or trace mode. This normally is not desirable since it slows down execution speed. To clear the breakpoint, use **be** with no arguments.

- **bm** Sets Memory Change Breakpoint

Syntax

bm [*range*]

Description

This command is used to set or clear memory change breakpoints. It does a checksum on the specified range on function entry and exit when the call trace mode is active (see **bt**), or on each instruction when the source line trace mode is active (see **bT**) or when running in single step or trace mode. The best way to locate a memory change is to first use the call trace mode to narrow it down between function calls. Then use the source line trace or single step mode to find the actual instruction.

When using single step mode in locating a memory change, enter the memory change breakpoint and then enter a large numeric prefix followed by an **S** or **T**. This will allow **sdb** to continue to check the location after each instruction and at the same time not require any input from you until the memory location has changed and the breakpoint activated or the specified number of instructions have been executed.

To clear a memory change breakpoint, enter **bm** with no range.

- **br** Resets breakpoint counters.

Syntax

br [*addr*]

Description

The **br** command resets the hit counter for the specified breakpoint which is at the address *addr*. If *addr* is not given, the hit counters for all breakpoints in the breakpoint table are reset.

- **bs** Set or modify a breakpoint

Syntax

[#] **bs** *addr* [*command*]

Description

The **bs** command enters a breakpoint into the breakpoint table, or modifies an existing entry.

The optional parameter **#** is the skip count for the breakpoint. If not specified, the skip count is set to 0, meaning that each time the breakpoint is reached, it will be taken.

The optional parameter *cmdlist* is a list of debugger commands to be executed when the breakpoint is taken.

- **bt** Toggle the call trace mode flag
- **bT** Toggle the source line trace mode flag

Syntax

bt
bT

Description

The **bt** and **bT** commands toggle the call trace mode and source line trace mode flags, respectively. The state of the trace mode flag determines whether trace mode is enabled or disabled.

In call trace mode, the debugger prints the names and arguments of each call within the program as it executes. On return, the value of the function's return is printed.

In source line trace mode, each statement of the program is displayed before it is executed.

DISPLAY COMMANDS

- **c** Display Source Context

Syntax

c

Description

This command centers the active source line in the source window.

- **da** Display local addresses

Syntax

da
dA

Description

The **da** command displays the automatic variables of the current function, while **dA** displays their addresses.

- **db** Display memory in bytes
- **dw** Display memory in words
- **dl** Display memory in long words
- **d** Display memory in last format

Syntax

db *[range]*
dw *[range]*
dl *[range]*

Description

The **db**, **dw**, and **dl** commands display successive bytes and words of memory, respectively.

d displays memory using the last format specified. For example, if **d** is entered, and **db** was the last display memory command, then **d** displays bytes, too.

The starting address of the *range* parameter is optional; if not specified, it defaults to the ending address of the last display's *range*, plus one. Each line of the display begins with the segment and address, followed by a hexadecimal display of 16 bytes or 8 words, followed by an ASCII display, by bytes, of the same data.

For the ASCII display, values falling outside the range 0x20 to 0x7f are displayed as a period.

If the ending address does not fall on a multiple of 16 bytes, only the number of bytes or words specified in the last line will be displayed.

- **dc** Display all code symbols

Syntax

dc

Description

The **dc** command lists all the code symbols in the memory-resident symbol table. For each symbol, its name and address are displayed.

- **dd** Display all data symbols

Syntax

dd

Description

The **dd** command lists all the data symbols in the memory-resident symbol table. For each symbol, its name and address are displayed.

- **df** Display source file lines

Syntax

df [*filename*] *range*

Description

The **df** command displays lines from the source file that was specified by the *filename* parameter.

The *range* parameter specifies the numbers of the lines to be displayed.

The starting line number is optional; if not specified, the display starts with the current line.

The current line in a source file is set by the source file commands **df** and **/**, as follows:

- When the file is first loaded with the **df** command, the first line in the file is the current line
- When the last source file command was **DISPLAY SOURCE**, **df**, the current line is the line following the last one displayed
- When the last source file command was **FIND STRING**, **/**, the current line is the line in which the string was found

The **df** command also sets the **F-DOT** for the source file to the number of the first line displayed. The **F-dot** is the line referred to when the starting line number of the range in a **df** command specifies a period (.). Also, source string searches begin at the line following the **F-dot** line.

Each displayed line is preceded with a line number in decimal, a colon, and the line itself.

➤ **dg** Display global values

Syntax

dg

Description

For each data symbol in the debugger's symbol table, **dg** displays the type, name, and value for that symbol. If the symbol is an array of structures, each element in the array will be printed.

- **ds** Displays stack backtrace
- **dS** Displays stack, function arguments, and auto backtrace

Syntax

[#] **ds**
[#] **dS** *command*

Description

The **ds** command displays information about the current function, the function that called it, etc., in reverse order of invocation, back to **__main()**, the Manx function which called the user's function **main()**.

The optional '#' parameter specifies the number of stack frames to be displayed.

For each function, the information consists of the function name and the parameters passed to it.

Arguments are displayed according to their type. If no type information is available, the arguments are displayed as a series of 16-bit hex values. Arguments of type **long** or **double**, when displayed as hex, will be displayed as separate words.

ds determines the number of parameters by looking at the instructions that follow the address to which the function returns.

ds assumes that the **a5** register points to the C stack frame for the current function, unless the current instruction is within 6 bytes of the start of the function.

dS causes the function to be displayed with the types, names, and values of each function argument. Below this, the values of all automatic variables are displayed.

"FIND SOURCE STRING" COMMAND

- **/** Finds string in source file

Syntax

/string

Description

This command searches the current source file (e.g., the one displayed in the source window) for a specified *string*.

The search begins at the line following the current line. If the *string* is found, the current line and the F-dot line of the source file is set to the line containing the *string*; otherwise, these values are unchanged.

string is the character *string* to be located, and consists of all characters following the / and preceding the carriage return.

If the first character of the *string* is ^, the search will begin with the first character on a line. In this case, ^ is not part of the search *string*.

The current line and F-dot line for a source file are defined in the description of the **df** command.

FRAME COMMANDS

- **fu** Frame up command
- **fd** Frame down command

Syntax

fu
fd

Description

Normally, you are only allowed to view variables that are visible by C rules at the point where execution stopped. That is, you cannot refer to local variables of functions that are not the current active function.

sdb allows you to get around this restraint by allowing you to change what the current function is.

The **fu** command allows you to walk up the call frame and displays the line from which the call was issued. By changing frames, you can view all the local variables of the now current function. You can walk up the frame until you get to the initial function.

The `fd` commands allows you to walk down the frame displaying the function call of the previous frame.

GO COMMANDS

- `g` Execute the program
- `G` Execute the program, without setting table breakpoints

Syntax

```
[#]g [addr] [command]
[#]G [addr] [command]
```

Description

The `g` commands transfer control of the processor to your program, at the address specified by the `pc` register. Your program then executes until it terminates, an error (such as division by zero) occurs, or a breakpoint is taken. Control then returns to the debugger program.

The parameters to the `g` commands allow one or two temporary breakpoints to be set in memory before your program is executed.

The difference between the `g` and the `G` command is that the `G` command sets in memory just the breakpoints specified in the command itself, while the `g` command also sets the breakpoints specified in the breakpoint table.

The `#` and `addr` parameters define one of the temporary breakpoints that a `go` command can set:

- `#` is the skip count for the breakpoint. It defaults to zero, meaning that the breakpoint is taken every time it is reached.
- `addr` is the address for the breakpoint.

The `@ <function>` parameter specifies that a temporary breakpoint is to be set at the return address of the specified function. If the function is not specified, it defaults to the current function. If a function is specified, the breakpoint is set to the address to which the function returns. In this case, the breakpoint is not set until the function is entered. Thus, in programs

that call the function from several different places, the breakpoint will be set at the actual address to which the function returns.

The *cmdlist* parameter defines a sequence of debugger commands, separated by semicolons, that the debugger is to execute once a breakpoint that is specified in the **go** command is taken. If this parameter is not specified, it defaults to the command list used for the last temporary breakpoint.

Before setting breakpoints and transferring control to your program, the debugger single steps your program (that is, causes it to execute one instruction). This allows the operator to transfer control to a location in the program at which there is a breakpoint, without immediately triggering a breakpoint and re-entry to the debugger.

LOAD COMMANDS

- **ld** load device symbols
- **ll** load library symbols
- **lp** load program

Syntax

```
ld devname
ll libname
lp
lp progfile [arg1 arg2 ...]
```

Description

The **ld** and **ll** commands are used to open a second **.dbg** file for debugging a library or device. (See the **ax** command as well as the **Debugging Libraries** or **Device Drivers** section of this chapter for a detailed description of these commands.)

The **lp** command loads a program into memory. If a symbol table file can be found for the program, it is loaded, too.

If the **lp** command is given without parameters, the last **lp** command is reexecuted. The following comments describe the parameterized version of **lp**.

Loading the Program

The parameter *progfile* specifies the file containing the program.

If the *progfile* specifies a drive or path, the file is searched for in just that location; otherwise, it is searched for on the current directory of the default drive.

If an attempt is made to load a program when *sdb* was invoked with a program name or a previous *lp progfile* was executed, an error will be printed and the command ignored.

Loading the Symbol Table

The name of the file containing the symbol table is assumed to be the same as the program filename, with the extension changed to *.dbg*.

After the program is loaded, its registers are initialized as though called from the CLI with a pointer to the program's arguments. These are available to the program as the main function arguments *argv[1]*, *argv[2]*, and so on. *argv[0]* is set to the name of the program being debugged, and *argc* is set to the number of *arg* parameters.

The *U-DOT* (e.g., the value of the period parameter associated with the *u* commands) is set to *Pc*. The *D-DOT* (the value of the period parameter for the *d* and *m* commands) is set to 0.

Once a program exits, it must be reloaded with an *lp* command before it can begin again.

MEMORY MODIFICATION COMMANDS

- **mb** Modify bytes of memory
- **mw** Modify words of memory
- **ml** Modify long words of memory

Syntax

mb	<i>addr</i>	<i>val1</i>	[<i>val2</i> ...]
mw	<i>addr</i>	<i>val1</i>	[<i>val2</i> ...]
ml	<i>addr</i>	<i>val1</i>	[<i>val2</i> ...]

Description

mb, **mw**, and **ml** modify bytes, words, and long words of memory, respectively. The parameter *addr* specifies the address of the first byte or word to be modified.

The *val* parameters are expressions whose resulting values are set in memory, with *val1* set in the first byte or word specified, *val2* set in the next higher byte or word, and so on.

The *val* parameters can be separated by spaces or commas.

- **mc** Compare memory

Syntax

mc *range* = *addr*

Description

The **mc** command compares two blocks of memory and, for each comparison that fails, displays the corresponding segment, address, and value.

range specifies one of the blocks of memory. The second begins at *addr* and has the same length as the first block.

- **mf** Fill memory with value

Syntax

mf *range* = *val*

Description

The **mf** command sets each byte in a block of memory to a specified value. The *range* parameter specifies the memory block, and *val* an expression whose resulting value is the value to be set in the range.

- **mm** Move memory

Syntax

mm *range* = *addr*

Description

The **mm** command copies one block of memory to another.

The *range* parameter specifies the source block and *addr* the starting address of the block to be modified.

- **ms** Search memory

Syntax

ms *range* = *val1* [*val2* ...]

Description

The **ms** command searches a block of memory for a sequence of bytes having specified values. For each match, the corresponding address of the start of the string is displayed.

range specifies the block of memory. The *val* parameters are expressions, each of whose resulting values is one byte of the search sequence.

OUTPUT CONTROL COMMANDS

- **oe** Toggles command echo flag

Syntax

oe

Description

Normally, commands typed into the command area are not displayed in the data area. This command acts as a toggle and causes most commands to be echoed in the data display. This is useful to separate the output of different commands.

- **op** Toggles pause flag

Syntax

op

Description

Normally, when more than one data window full of output is generated, **sdb** prompts

<Hit Any Key to Continue>.

This command acts as a toggle, allowing the output to continue scrolling even though it contains more than one window full of information.

- **ow** Toggles switch to user screen

Syntax

ow {on | off}

Description

Normally, **sdb** only swaps the application screen to the front when you type **g**, **s**, or **t**. This is done to avoid unnecessary screen flicker when single stepping through a program. The only problem is when the application modifies the screen. If this happens, the **sdb** screen is altered unless it has been swapped with your screen. Using this command allows screen swapping to be switched on/off.

PRINT COMMAND

- **p** Generates formatted output

Syntax

P [*format*] [*range*] [,*next*]

Description

The **p** command generates a formatted display of C variables, arrays, and structures, by converting data items in memory to a displayable format directed by its type or the optional conversion string format.

format is an optional list of format specifications, each of which defines the type of a data item and the conversion to be performed on it.

addr specifies the address of the first data item that **p** is to convert and display.

In the absence of the format string, **p** looks at the *addr* to be printed, gets its typing information from the symbol table, builds and executes the appropriate format string. For example, to print a structure named symbol you simply enter:

```
p symbol
```

which might result in:

```
struct symboltb symbol = {  
    int s_flag = 10  
    char *s_name = 0xFF42  
    int s_value[2] = {  
        10,5  
    }  
}
```

If you then want to display the string pointed to by **symbol.s_name**, you simply type:

```
ps *symbol.s_name
```

The **s** indicates that you are overriding the default format for pointer to char. The result might be:

```
pointer
```

If you wish to create your own format string, here is a description of the use of format by **p**: **p** works its way through the format string, converting and displaying data items in memory as requested by the format string items. When **p** reaches an item in the format string, it converts the data item at its current address as directed by the format item. When it finishes processing a format string item, it increments its current address by the size of the data item that it just processed to be ready to process the next data item as directed by the next format string item.

If *addr* is not entered, the starting address is assumed to be the print command's current address.

Normally, this is the address of the first byte beyond the last data item converted by the last **p** command. However, there is a format item that causes **p** to remember the address contained in the current data item and then make that the current address after it finishes processing the entire format string.

The format items have the form

[*rpt*] [*indir*] [*size*] *desc_code*

where

desc_code is a single-letter code that defines the type of the data item and the conversion to be performed upon it. For example, the code **d** says “take the two-byte binary value at the current address, convert it to decimal, and print it”. Therefore, if **var** is an **int**, the following command could be used to print its value in decimal:

pd var

The code **x** says “take the two-byte binary value at the current address, convert it to hexadecimal, and print the result”. So the hexadecimal value of **var** could be printed with the command:

px var

A format item’s *indir* is a string that consists of zero or more ***** characters that are indirection indicators specifying that the value at the current data item is a pointer to a chain of zero or more pointers, the last of which points to an object whose type and requested conversion are defined by *desc_code*.

To find the data object corresponding to a format item that has indirection indicators, **p** begins by setting its idea of the address of the data object to the current address. It then works its way from left to right through the indirection indicators; for each indicator it replaces its current idea of the data object address with the pointer that is in the field at this address. The data object address is distinct from the current address: At the end of this process, the **p** command’s current address is simply incremented past the first pointer.

For example, if the variable **cp** is a short pointer to a character string (e.g., its declaration is **char *cp**), then the string pointed at by **cp** could be printed by the command:

p*s cp

Here we have used the **s** *desc_code*, which specifies that the data object is a character string, and that the string’s characters are to be printed, with possible modifications as noted below, up to a terminating null character.

After this command, the **p** command's current address is set to the byte immediately following **cp**.

The *rpt* parameter of a format item defines the number of times that the item is to be processed. It allows a sequence of *rpt* identical format items to be abbreviated by just one such item with a leading *rpt* count. For example, if **a** is an array of floats, then the first five items in this array could be displayed with the command

```
p5f a
```

This command uses the fact that the *desc_code* to convert a four-byte floating point value at the current address to a displayable value is **f**. This command is equivalent to the command **pf f f f f a**. At the end of this command, the **p** command's current address is set to the address of the byte following the last displayed float.

The *size* parameter of a format item defines the number of data items that are to be converted and printed. When the format item does not use indirection, *size* has the same effect as *rpt*; for example, in the **p5f a** command above, the **5** could be interpreted as being a *size* parameter instead of an *rpt* parameter.

When the format item does use indirection, then the *size* parameter defines the number of data items to be converted and printed at the end of the indirection chain. For example, if a module defines **lpp** as a pointer to an array of pointers to longs (e.g., the declaration of **lpp** is **long **lpp**), then the following command would display the first four longs pointed at by the first element of the pointer array:

```
p**4D lpp
```

Here we have used the *desc_code* **D**, which specifies that a four-byte signed binary value is converted to decimal and printed.

The following command would display the first three longs pointed at by the first three elements of the pointer array:

```
p3*4D *lpp
```

To demonstrate further the difference between the *rpt* and *size* fields in a format item, consider the format items **4*d *4d**. The first causes the print command to take the item at the current address as a pointer, increment the current address by two, convert to decimal and print the two-byte value

referenced by the pointer, and then repeat the process three more times. At the end of the process, the current address has been advanced by eight.

The second item causes the print command to again take the item at the current address as a short pointer, increment the current address by two, and then convert to decimal and print the four successive two-byte values that begin at the address defined by the pointer. At the end of the process, the current address has been advanced by two.

As an example of the use of format strings containing several format items, consider the following code:

```
struct {  
    int *ip;  
    float flt;  
    char *cp;  
} var = (&i, 3.14159, "ralph");  
int i=2;
```

The command:

```
p*d2-xf*s2-x var
```

prints:

```
2 xxxx 3.14159 "ralph" yyyy
```

where **xxxx** is the hexadecimal address of **i** and **yyyy** is the hexadecimal address of the string.

desc_code List

The basic *desc_codes* are as follows:

- | | |
|----------|---|
| b | Converts to hexadecimal and prints a byte. |
| d | Converts to decimal and prints a two-byte signed binary value. |
| D | Converts to decimal and prints a four-byte signed binary value. |
| e | Converts and prints a long double floating point value |

- f** Converts and prints a double floating point value.
- F** Converts and prints an eight-byte double.
- .** Defines the precision to be used in the display of floating point values, e.g., the number of digits to be displayed to the right of the decimal point. This number precedes the period. For more information, see below.
- o** Converts to octal and prints a two-byte field.
- O** Converts to octal and prints an eight-byte field.
- x** Converts to hexadecimal and prints a 2-byte field.
- X** Converts to hexadecimal and prints a 4-byte field.
- u** Converts to decimal and prints an unsigned, two-byte value.
- U** Converts to decimal and prints an unsigned, four-byte value.
- c** Prints a character.
- C** Prints a character.
- s** Prints a string up to a terminating null byte.
- S** Prints a string with (without) translation.

Floating Point Number Print Codes

When a floating point number is displayed in response to an **f** or **F** *desc_code*, the precision of the display (e.g., the number of digits displayed to the right of the decimal point) is determined by the most recently-entered period *desc_code*. If a period code has not been entered, the precision of the display defaults to 7 digits for floats and 16 for doubles.

The period *desc_code* consists of a period preceded by the number of digits of precision. For example, the following command contains two *desc_codes*. The first sets the precision to 2 digits, the second then displays a float, listing two digits to the right of the decimal point:

p2.F

Character Print Codes

For the **C** and **S** *desc_codes*, each character is printed as is, with no translations. For the **c** and **s** codes, printable ASCII characters (e.g., whose hex value is between 0x20 and 0x7f) are printed as is. A character whose hex value is less than 0x20 is printed as two characters: ^ followed by the printable character whose hex value equals the original character's value plus 0x40. A character whose hex value is 0x80 or greater is displayed as a ' character followed by the one or two characters that would be printed for the character whose hex value equals that of the original character less 0x80. For example, 0x41, 0x1, and 0x81 would be printed as A, ^A, and '^A, respectively.

Special Purpose Codes

The following *desc_codes* can be used to assist in the formatting of the **p** output:

Character	Output
N or n	Outputs a newline character
R or r	Prints a name
T or t	Outputs a tab character
"string"	Outputs "string"

These characters can be preceded by a count specifying the number of characters or strings to be output.

P Command *desc_codes* For Setting Current Address

The next group of *desc_codes* change the **p** command's notion of the current address. They do not cause any printing.

- ^** Backs up the current address by the size of the last data item.
- or +** Backs up or advances, respectively, the current address by *size* bytes, where *size* is a decimal value preceding the - code. If *size* is not specified, it defaults to one byte.
- A or a** Remembers the long or short pointer, respectively, that is contained in the current data object. If this pointer is not null, sets the **p** command current address to this value after the entire format string has been processed.

If the pointer is null, sets the **p** command's current address to the value it had before the entire format string was processed.

The **A** and **a** *desc_codes* are useful for printing the elements of a linked list. For example, consider the following code, which defines the structure for a symbol table item and declares **sym_head** to be a pointer to this structure. The program that uses this structure and field will chain symbol table items together and set a pointer to the head of the chain in **sym_head**.

```
struct symbol {  
    struct symbol * sym_next;  
    char *sym_name;  
    unsigned sym_val;  
    *sym_head;
```

The following command would display the symbol table item pointed at by **sym_head** and then set the **p** command's current address to the next symbol table item, which is pointed at by the **sym_next** field in the first item:

```
p A"symbol name="#snt"value="x sym_head
```

After this command is entered, you can display successive symbol table items by simply entering

P

The **p** command's current address is correctly set to the next table item and since a format string is not specified, the **p** command uses the one that it last used.

You can print out multiple symbol table items by entering a single **p** command. To do this, place a comma and the maximum number of items to be printed after the command's starting address. The command follows the chain, printing symbol table items until it either prints the specified number of items or it prints an item whose **sym_next** pointer is null. In the latter case, it will terminate and leave the **p** command's current address set to the address of the last symbol table item. For example, entering

```
p A"symbol name="#snt"value="x sym_head,100
```

prints symbol table items until it either prints 100 items or it prints an item having a null **sym_next** pointer.

If **P** is used, the address of the item is displayed as well.

.QUIT COMMAND

► **q** Quit the debugger

Syntax

q

Description

sdb terminates the program being debugged by first checking for an **_abort()** code symbol or an **exit()** code symbol. If found, the program counter is set to that symbol and the program is allowed to resume.

This is the normal way to exit **sdb** under most circumstances.

REGISTER COMMAND

- **r** Register display

Syntax

r
r <reg>=val

Description

The **r** command displays and modifies the registers, including the status registers, of the program being debugged.

The parameter-less version displays the registers. The parameterized version modifies the contents of a register, with <reg> being the name of the register to be modified, and *val* an expression whose resulting value is to be set into the register.

If a MC68881 floating point chip is present and not in the reset state, its registers will be displayed as well.

SINGLE STEP COMMANDS

- **s** Single steps the program *n times*
➤ **S** Single steps into calls without display
➤ **t** Single steps *n times* stepping over calls
➤ **T** Single steps over calls, display last

Syntax

#s
#S
#t
#T

Description

These commands SINGLE STEP your program, e.g., execute its instructions one by one.

The optional **#** parameter specifies the number of instructions to be executed. It defaults to one instruction.

If the count is zero, it goes forever, e.g., **OT** single steps until a breakpoint or an error is encountered.

The **s** and **S** commands differ in that **s** displays information after each single step, whereas **S** only displays information after the last single step.

The same is true for the **t** and **T** commands.

The **s** and **t** commands differ in that the **s** commands single step into calls when encountered while the **t** command treats a function call as a single step and steps over it.

The displayed information consists of the source line of the next instruction to be executed. When single stepping, breakpoints are not enabled.

UNASSEMBLE COMMANDS

- **u** Unassembles memory, with symbols
- **U** Unassembles memory, without symbols

Syntax

u *range*
U *range*

Description

These commands **DISASSEMBLE** a *range* of memory, e.g., display the assembly language instructions in the *range*.

Both the **u** and **U** commands make use of the symbol table during disassembly. They differ in that the **u** command includes the symbols+offset as part of the assembly language while the **U** command prints locations in hex with the symbol and offset for the location displayed at the far right of the line. Also, the **U** command displays, for each instruction, the hex value of each byte of the instruction, whereas the **u** command will not.

With the **u** command, the disassembly of an instruction that references memory displays the location as the symbol nearest to the location plus an

offset, if possible. With the **U** command, the location is displayed as a hexadecimal value.

The *range* parameter specifies the area of memory to be disassembled. It gives the starting address and either the number of instructions to be disassembled or the ending address of the area.

Only the **u** (lowercase) command displays the source lines and autos as comments.

MACRO COMMANDS

- **x** Create Macro Command
- **X** Display Macro Command

Syntax

x name macro
X name

Description

Macro The **x** command defines or executes a sequence of debugger commands, called a macro. **X** lists the defined macros.

A *macro name* consists of a sequence of alphanumeric characters, the first of which must be alphabetic, with a limit of 40 characters. Case is not significant.

A macro is defined by typing the letter **x**, followed by the *name* with which the macro is to be associated. Then follows the macro's list of debugger commands with the commands separated by semicolons.

A macro is executed by typing **x**, followed by the *name* with which the macro is associated, followed by a carriage return.

The macros that have been defined can be listed using the command **X**.

Macros are automatically saved in a file called *xxx.mac* where *xxx* is the same as the name of the program being debugged. This file is executed automatically when **sdb** is started.

EXPRESSION COMMANDS

- **=** Display the value of an expression in several formats
- **e** Evaluate an expression

Syntax

= *expr*
e *expr*

Description

The **=** command displays the value of an expression.

The expression is displayed in several formats: hexadecimal, signed decimal, unsigned decimal, octal, binary, and ASCII. If a symbol table has been loaded, the closest symbol is displayed as well.

The **e** command allows you to perform C expression evaluation, including calls to C functions, assignment, pre- and post- increment and decrement, casts, and conditionals. For example:

```
e c = getchar()
```

might result in:

```
=10
```

REDIRECT COMMAND INPUT/OUTPUT

- **<** Redirect command input
- **>** Log all I/O only
- **>>** Log commands only

Syntax

<*filename*
>*filename*
>>*filename*

Description

The < command causes the debugger to redirect input from a file. This command provides a convenient means for defining macros or variables.

When the end of the file is reached, the debugger returns to the console for commands.

The > command causes the debugger to copy all output to the specified file. This feature allows you to keep a record of the commands and responses for a debugging session.

The >> command causes the debugger to log commands only to the specified file. This is useful if you wish to reexecute a sequence of commands to reproduce a problem.

HELP COMMAND

➤ ? List command

Syntax

?

Description

This command lists the debugger commands. For groups of related commands, the listing usually lists the first letter of the commands followed by a ?. You can get a listing of all the commands in such a group by typing the letter, the ?, and return.

For example, the listing for the display commands is d?. Thus you can type d? followed by return to get a listing of all the display commands.

CHANGE MODE COMMAND

➤ z Change mode command

Syntax

z

Description

The **z** command allows you to switch from source mode to assembly mode and back again. By default, the debugger starts in source mode unless you specify option **-a** on the command line.

DELAY COMMAND

► **&** Delay **sdb**

Syntax

& [*cnt*]

Description

This command delays **sdb** for *cnt* seconds or until you hit return if a *cnt* is not given. Its primary purpose is for use in the demo scripts where the application screen is displayed for a couple of seconds while **sdb** waits.

Command Summary

AMIGA SPECIFIC COMMANDS

acc	show/set the command window colors
acd	show/set the data window colors
acs	show/set the source window colors
add	display device list
adi	display interrupt list
adl	display library list
adp	display port list
adr	display resource list
am	display memory usage
ax	switch between main and alternate .dbg file

BREAKPOINT COMMANDS

bc/bC	clear one/all breakpoints
bd	display the breakpoint table
be	set expression breakpoint
bm	set memory change breakpoint
br	reset the breakpoint counters
bs	set or modify a breakpoint
bt/bT	enable/disable trace mode

DISPLAY COMMANDS

c	display source context
da/dA	display local variables/addresses
db/dw/dl/d	display memory in bytes/words/long words/last format
dc/dd	display code/data symbols
df	display source file lines
dg	display global values
ds/dS	display stack backtrace

FIND SOURCE STRING COMMAND

/ find string in source file

CHANGE FRAME COMMANDS

fu walk up the call frame

fd walk down the call frame

GO COMMAND

g/G execute program

LOAD COMMAND

ld load device symbols

ll load library symbols

lp load program

MEMORY MODIFICATION COMMANDS

mb/mw/ml modify bytes/words/long words of memory

mc compare areas of memory

mf fill memory

mm move memory

ms search memory

OUTPUT CONTROL COMMANDS

oe toggle command echo flag

op toggle pause flag

ow toggle switch to user screen

FORMATTED PRINT COMMAND

p generate formatted print

QUIT COMMAND

q quit debugger

REGISTER COMMAND

r register display

SINGLE STEP COMMANDS

s/S single step with/without display, stepping into function calls

t/T single step with/without display, stepping over function calls

UNASSEMBLY COMMANDS

u/U unassemble memory

MACRO COMMANDS

x/X define or execute/display a command macro

EXPRESSION COMMANDS

= display value of an expression in several formats

e evaluate an expression

REDIRECT COMMAND INPUT

< redirect command input

> log all input/output

>> log commands only

HELP COMMAND

? list debugger commands

CHANGE MODE COMMAND

z switch from source to assembly mode and back

DELAY COMMAND

& delay until return or specified seconds

Index

- #acs command, 4-46
- \$ character, 4-8
- & command, 4-45, 4-48
- a option, 2-4
- cc option, 2-4
- cd option, 2-4
- cs option, 2-4
- cw option, 2-5
- d option, 2-5
- e option, 2-5
- h option, 2-5
- m option, 2-5
- n option, 2-5
- p option
 - w mode, 2-5
- r option, 2-5
 - use with REZ utility, 2-5
- s option, 2-5
- w option, 2-5
- / command, 4-7, 4-25, 4-47
- 68010, 1-2
- 68020, 1-2
- 68030, 1-2
- < command, 4-11, 4-44
 - See also redirect commands
- = command, 4-11, 4-43, 4-48
 - expression commands, 4-43
- > command, 4-11, 4-44
 - See also redirect commands
- >> command, 4-11, 4-44
 - See also redirect commands
- ? command, 4-2, 4-11, 4-44, 4-48

A

- acc command, 4-15, 4-46
 - description, 4-15
- acd command, 4-15, 4-46
- acs command, 4-15 - 4-16
- add command, 4-4, 4-16, 4-46
- ADDR, 4-14
 - definition, 4-13
 - examples, 4-13
- adi command, 4-16, 4-46
- adl command, 4-4, 4-16, 4-46
- adp command, 4-16, 4-46
- adr command, 4-16, 4-46
- am command, 4-17, 4-46
- Amiga specific commands
 - listing, 4-46
 - acc command, 4-15
 - acd command, 4-15
 - acs command, 4-15 - 4-16
 - add command, 4-16
 - adi command, 4-16
 - adl command, 4-16
 - adp command, 4-16
 - adr command, 4-16
 - am command, 4-17
 - ax command, 4-17
- ax command, 4-4, 4-17, 4-46

B

- b commands, 4-5
- backtracing, 4-6
- basic commands
 - display, 4-1
 - evaluate, 4-2
 - frame, 4-2
 - help, 4-2
 - memory modify, 4-2
 - print, 4-2
 - register, 4-2

- single step, 4-2
- transfer control commands, 4-2
- unassemble, 4-2
- bc/bC command, 4-17, 4-46
- bd command, 4-18, 4-46
- be command, 4-6, 4-18, 4-46
- bin directory, 2-2
- bm command, 4-19, 4-46
- br command, 4-20, 4-46
- breakpoint commands, 4-5
 - See also specific command
 - b, 4-17 - 4-18
 - bc/bC, 4-17
 - bd, 4-18
 - be, 4-18
 - bm, 4-19
 - br, 4-20
 - bs, 3-4, 4-20
 - bt/bT, 4-20
 - listing, 4-46
- breakpoints, 4-5 - 4-6
 - expression, 4-18
 - memory change, 4-19
 - one-time, 3-3
 - permanent, 3-3 - 3-4
- bs command, 4-20, 4-46
 - examples, 3-4
- bt/bT command, 4-6, 4-20, 4-46

C

- c command, 3-2, 4-7, 4-21, 4-46
- C source file, 2-3
- call trace mode, 4-20
 - enabling and disabling, 4-6
- categories
 - backtracing, 4-6
 - basic commands, 4-1 - 4-2
 - breakpoints, 4-5 - 4-6
 - command display, 4-9
 - command line, 4-10

- command line editing, 4-9 - 4-10
- debugging device drivers, 4-4
- debugging libraries, 4-4
- displaying source files, 4-7
- expression breakpoints, 4-6
- loading programs and symbols, 4-3
- macros, 4-7
- names, 4-3
- other features, 4-11 - 4-12
- source and data displays, 4-8
- term definitions, 4-12 - 4-14
- trace mode, 4-6
- windows, 4-7 - 4-8
- cd option, 2-4
- change frame commands
 - listing, 4-47
- change mode command
 - See z command
 - listing, 4-48
 - z, 4-45
- character print codes, 4-37
- CLI, 2-2
- CMDLIST
 - definition, 4-14
- code and data symbols, 4-3
- color control, 4-10
 - window colors, 2-5
- command description
 - general, 4-1
- command display, 4-9
- command input
 - redirection, 4-11
- command line editing, 4-9 - 4-10
 - ^C, 4-10
 - ^X, 4-10
 - ESC character, 4-10
 - F10 function key, 4-10
- command summary
 - amiga specific commands, 4-46
 - breakpoint commands, 4-46

- change frame commands, 4-47
- change mode command, 4-48
- delay command, 4-48
- display commands, 4-46
- expression commands, 4-48
- find source string command, 4-47
- formatted print command, 4-47
- go command, 4-47
- help command, 4-48
- load command, 4-47
- macintosh specific commands, 4-46
- macro command, 4-48
- memory modification commands, 4-47
- output control commands, 4-47
- quit command, 4-47
- redirect command input, 4-48
- register command, 4-48
- single step commands, 4-48
- unassembly commands, 4-48
- commands, detailed descriptions
 - See also SDB commands
 - Amiga specific, 4-14
 - breakpoint, 4-17 - 4-19
 - change mode, 4-45
 - delay, 4-45
 - display, 4-21 - 4-24
 - expression, 4-19, 4-43
 - find source string, 4-25
 - frame, 4-25
 - go, 4-26
 - help, 4-44
 - load, 4-27
 - macro, 4-42
 - memory modification, 4-28 - 4-29
 - output control, 4-30
 - print, 4-31 - 4-36, 4-38 - 4-39
 - quit, 4-39
 - redirect input/output, 4-44
 - register, 4-40
 - single step, 4-40 - 4-41

- unassemble, 4-41 - 4-42
- compare memory command, 4-29
- compiler
 - bs option, 2-3, 4-4
- control characters
 - ^, 4-12
 - ^C, 4-10
 - ^D, 4-12
 - ^G, 4-12
 - ^K, 4-12
 - ^L, 4-12
 - ^N, 4-12
 - ^T, 4-12
 - ^U, 4-12
 - ^V, 4-12
 - ^W, 4-12
 - ^X, 4-10
 - ^Y, 4-12
- control characters, use of, 4-12
- count, 3-2
- loading sdb
 - creating "work disk", 2-2
- cs option, 2-4
- current source line, 3-2
- cw option, 2-5

D

- d command, 4-5, 4-21, 4-46
- da/dA command, 4-21, 4-46
- db command, 4-21, 4-46
- dbg file command
 - switch between main and alternate, 4-5
- dc command, 4-22, 4-46
- dd command, 4-22, 4-46
- debug file, 2-3
- debugging device drivers, 4-4
- debugging libraries, 4-4
- default settings, 4-11
- delay command
 - &, 4-45

- listing, 4-48
- demonstration programs, 2-1, 2-3
- desc_code list, 4-35 - 4-38
- desc_codes
 - character print codes, 4-37
 - examples, 4-33 - 4-34, 4-38
 - floating point number print codes, 4-36
- device drivers
 - debugging, 4-4
- df command, 3-2, 4-7, 4-22 - 4-23, 4-46
 - range, 3-2
- dg command, 4-23, 4-46
- disassembly
 - See unassemble commands
- display code command, 4-3
- display code symbols, 4-22
- display command, 4-5
- display commands, 4-2
 - See also specific commands
 - =, 4-43
 - add, 4-16
 - adi, 4-16
 - adl, 4-16
 - adp, 4-16
 - adr, 4-16
 - am, 4-17
 - bd, 4-18
 - c, 4-21
 - d, 4-21
 - da/dA, 4-21
 - db, 4-21
 - dc, 4-22
 - dd, 4-22
 - df, 4-22 - 4-23
 - dg, 4-23
 - dl, 4-21
 - ds/dS, 4-24
 - dw, 4-21
 - listing, 4-46
 - r, 4-40

- s/S, 4-40
- t/T, 4-40
- x/X, 4-42
- display context command, 4-21
- display data symbols, 4-22
- display global value, 4-23
- display memory commands, 4-21
- display register command, 4-40
- display source file lines, 4-22
- display symbols command, 4-3
- displaying assembly, 3-7
- displaying information
 - current function, 4-7
- displaying source files, 3-2, 4-7
- displaying the trace of calls, 3-5
- displaying values and computing expressions, 3-6
- displaying variables, 3-6
- dl command, 4-21, 4-46
- ds/dS command, 3-5, 4-7, 4-24, 4-46
 - example, 3-5
- dw command, 4-21, 4-46

E

- e command, 3-6, 4-2, 4-43, 4-48
 - example, 3-6
- environment variables
 - SDBOPT, 2-5, 4-11
- evaluate command, 3-6, 4-2
- evaluate expression command, 4-12
- execute program commands, 4-26
- execution
 - controlling, 3-3
- EXPR
 - definition, 4-13
 - example, 4-13
- expression breakpoints, 4-6
- expression commands
 - =, 4-43
 - e, 4-43
 - listing, 4-48

F

- f command, 4-2
- fd command, 3-7, 4-25, 4-47
- file preparation
 - bs compiler option, 2-3
 - g linker option, 2-3
 - executable files, 2-3
- files for debugging
 - preparation of, 2-1
- find source string commands, 4-7
 - /, 4-25
 - listing, 4-47
- floating point number print codes, 4-36
- format items
 - desc_codes, 4-34 - 4-35
 - examples, 4-34 - 4-35
- formatted print command
 - listing, 4-47
- frame commands, 4-2
 - fd, 3-7, 4-25
 - fu, 3-7, 4-25
- fu command, 3-7, 4-25, 4-47

G

- g/G command, 3-4, 4-2, 4-5, 4-26, 4-47
- go command, 3-4
 - See also g/G command
 - controlling execution, 3-3
 - g/G, 4-26
 - listing, 4-47

H

- help command, 4-2, 4-12
 - ?, 4-44
 - listing, 4-48
- help screens, 3-1

I

- installation, 2-2
 - getting started, 2-1
 - using floppy disk, 2-2
 - using hard disk, 2-2

K

- keys
 - See also control characters

L

- ld command, 4-4, 4-27, 4-47
- line, 3-2
- linker
 - g option, 2-3, 4-3 - 4-4
- list command
 - See help command
- ll command, 4-4, 4-27, 4-47
- load command
 - description, 4-27
 - ld, 4-27
 - listing, 4-47
 - ll, 4-27
 - loading the program, 4-28
 - loading the symbol table, 4-28
 - lp, 4-27 - 4-28
- load device symbols command, 4-27
- load library symbols command, 4-27
- load program command, 4-3, 4-27
- loading programs and symbols, 4-3
- loading sdb
 - using floppy drives, 2-2
 - using hard disk, 2-2
- log commands, 4-11
 - <, 4-43
 - >, 4-43
 - >>, 4-43
- lp command, 2-4, 4-3, 4-27 - 4-28, 4-47

M

- macro commands
 - listing, 4-48
 - x/X, 4-42
- macros
 - defining and executing, 4-7
 - fpath, 2-6
- mb command, 4-28, 4-47
- mc command, 4-29, 4-47
- memory modification commands, 4-2
 - listing, 4-47
 - mb, 4-28
 - mc, 4-29
 - mf, 4-29
 - ml, 4-28
 - mm, 4-30
 - ms, 4-30
 - mw, 4-28
- mf command, 4-29, 4-47
- ml command, 4-28, 4-47
- mm command, 4-30, 4-47
- mode commands
 - See change mode command
- move memory command, 4-29
- ms command, 4-30, 4-47
- mw command, 4-28, 4-47

N

- names
 - code and data symbols, 4-3
 - operator usage of, 4-3
- notation conventions, 1-2

O

- oe command, 4-30, 4-47
- op command, 4-31, 4-47
- operator usage of names, 4-3
- options
 - a, 2-4

- cc, 2-4
- cd, 2-4
- cs, 2-4
- cw, 2-5
- d, 2-5
- e, 2-5
- h, 2-5
- m, 2-5
- n, 2-5
- p, 2-5
- r, 2-5
- s, 2-5
- w, 2-5
- options list, 2-5
- other features
 - control characters, use of, 4-12
 - evaluate expression command, 4-11
 - help command, 4-11
 - redirect input command, 4-11
- output commands
 - listing, 4-47
- output control commands
 - oe, 4-30
 - op, 4-31
 - ow, 4-31
- ow command, 4-31, 4-47

P

- p command, 4-2, 4-31 - 4-39, 4-47
 - current address setting codes, 4-38 - 4-39
 - example, 3-6
- parameters, 2-5
- Preferences program, 4-10
- preparation of files for debugging, 2-1, 2-3
- print command, 3-6, 4-2
 - character print codes, 4-37
 - current address setting codes, 4-38 - 4-39
 - desc_code list, 4-35 - 4-38
 - description, 4-31 - 4-34
 - examples, 4-32 - 4-34

- floating point number print codes, 4-36
- format items, 4-33, 4-35
- p, 4-31 - 4-39
- special purpose codes, 4-37
- processors supported, 1-2
- program
 - running, 3-3 - 3-4

Q

- q command, 4-39, 4-47
- quit command
 - listing, 4-47
- q, 4-39

R

- r command, 4-2, 4-40, 4-48
- range, 3-2, 4-14
 - definition, 4-13
- installation, 2-1
- README file, 1-2, 2-1
- redirect command input
 - listing, 4-48
- redirect commands, 4-44
 - <, 4-43
 - description, 4-44
- redirection, 4-11
- register command, 4-2
 - listing, 4-48
 - r, 4-40
- requirements, 1-2
- running the program, 3-3 - 3-4

S

- s option
 - fpath macro, 2-6
- s/S command, 3-3, 4-2, 4-40 - 4-41, 4-48
- scratch directory, 2-2
- SDB
 - definition, 1-1

SDB commands, 3-7

See also specific command

&, 4-45

/, 4-7, 4-24

<, 4-11, 4-43

=, 4-43

>, 4-11, 4-43

>>, 4-11

?, 4-44

ac, 4-11

acc, 4-15

acd, 4-15

acs, 4-15

add, 4-4, 4-16

adi, 4-16

adl, 4-4, 4-16

adp, 4-16

adr, 4-16

am, 4-17

ax, 4-17

b, 4-5

bc/bC, 4-17

bd, 4-18

be, 4-6, 4-18

bm, 4-19

br, 4-19

bs, 3-4, 4-20

bt/bT, 4-6, 4-20

c, 3-2, 4-7, 4-21

d, 4-5, 4-21

da/dA, 4-21

db, 4-1, 4-21

dc, 4-3, 4-22

dd, 4-3, 4-22

df, 3-2, 4-7, 4-22 - 4-23

dg, 4-23

dl, 4-1, 4-21

ds/dS, 3-5, 4-6, 4-24

dw, 4-1, 4-21

e, 3-6, 4-2, 4-43

- f, 4-2
- fd, 3-7, 4-25
- fu, 4-25
- g/G, 3-4, 4-2, 4-6, 4-26 - 4-27
- h, 4-2
- ld, 4-4, 4-27
- ll, 4-4, 4-27
- lp, 2-6, 4-3, 4-27 - 4-28
- m, 4-2
- mb, 4-28
- mc, 4-29
- mf, 4-29
- ml, 4-28
- mm, 4-29
- ms, 4-30
- mw, 4-28
- oe, 4-30
- op, 4-30 - 4-31
- ow, 4-31
- p, 4-1 - 4-2, 4-31 - 4-39
- p, 3-6
- q, 4-39
- r, 4-2, 4-40
- s/S, 4-2, 4-6, 4-40
- t/T, 3-3, 4-2, 4-40
- u/U, 3-7, 4-2, 4-41
- x/X, 4-7, 4-42
- z, 4-44
- SDBOPT environment variable, 2-5, 4-11
- search memory command, 4-30
- single step command
 - s/S, 3-3
- single step commands, 4-6
 - controlling execution, 3-3
 - default settings, 3-3
 - listing, 4-48
 - s/S, 4-40 - 4-41
 - t/T, 4-40 - 4-41
- sizing icon, 4-7
- source and data displays, 4-8

- source file, displaying sections, 3-2
- source line trace mode
 - enabling and disabling, 4-6
- source window, 3-2
- special purpose codes, 4-37
- startup, 2-6
- startup SDB, 2-1, 2-4
 - from Shell or CLI, 2-6
 - from the Aztec Shell, 2-4
- statics, 3-7

T

- t/T command, 3-3, 4-2, 4-40 - 4-41, 4-48
- technical support, 1-3
- term definitions
 - ADDR, 4-13
 - CMDLIST, 4-14
 - EXPR, 4-12
 - RANGE, 4-13
- trace mode, 4-6
- trace mode commands, 4-6
 - bt/bT, 4-20
- transfer control commands, 4-2, 4-5
- truncated lines, 4-8
- tutorial
 - introduction, 3-1 - 3-2

U

- u/U command, 3-7, 4-41 - 4-42, 4-48
 - See also unassemble commands
- unassemble commands, 4-2
 - disassembly, 3-7
 - listing, 4-48
 - u/U, 4-41 - 4-42

W

- walking up and down the frames, 3-7
- windows, 4-7
 - color control, 2-5

- colors, 2-5, 4-10
- source and data display, 4-8
- Workbench colors, 4-10

X

- x/X command, 4-7, 4-42, 4-48

Z

- z command, 3-3, 4-45, 4-48

